

编译技术第二次project报告

设计思路

主要分为两部分，第一部分是根据链式法则，定制Mutator，对在第一个project生成的IRTree进行节点的修改，生成新的IRTree。第二部分是使用第一次project中的IRPrinter打印生成代码。

第一部分

在查阅资料后，我们所采取的思路是借鉴反向自动微分法，所要做的就是求出IRTree中每个 `Expr` 的节点的梯度：从IRTree的根部开始，根据链锁法则，通过深度优先搜索的方式将根结点的梯度逐步向子节点传递，直到叶子节点。整个过程是从上至下的递归过程。

递归公式

梯度公式是算法的核心，梯度公式如下：

父亲节点为： $ExprC = ExprA$ ，则 $dA = dC$

父亲节点为： $ExprC = ExprA + ExprB$ ，则 $dA = dC$ ， $dB = dC$

父亲节点为： $ExprC = ExprA - ExprB$ ，则 $dA = dC$ ， $dB = -dC$

父亲节点为： $ExprC = ExprA \times ExprB$ ，则 $dA = dC \times B$ ， $dB = dC \times A$

在这种方式下，若出现新的运算符号，我们只需增加定义其递归公式即可。

下标处理

对于下标，我们在观察结果归纳后发现，下标的索引不需要做改变，只需保留即可。

代码实现方式

仍旧采用第一次project中的 `json_To_IRTree` 类来完成json到IRTree的转换。根据例子来看，这次不需要根据爱因斯坦求和范式来拆分式子，于是把所有temp数组去掉，并且只生成和 `grad_to` 的变量个数一样的多的move语句（放在循环最内部）。因为样例比较简单，我们把确保循环范围正确的if语句也去掉了。

对于json中的 `grad_to` 部分，直接解析其中的变量名并放入 `std::vector<std::string> grads` 中供后续使用。

针对 `json_To_IRTree` 生成好的kernel，mutator会对其进行一系列变换，具体如下。

0.mutator类中的变量

```

1   std::map<std::string, Expr> gradForVar;
2   Expr leftVar;
3   std::set<std::string> ins;
4   std::vector<Expr> gradsVar;
5   int moveNum=0;
6   bool moveRight=false;

```

gradForVar: 变量名-对应的梯度表达式

leftVar: 顶部节点的梯度表达式

ins: 新的kernel的ins部分, 也就是求导后等式右端出现的所有变量名

gradsVar: 需要求导的变量 (存的是Expr, 其实是Var)

moveNum: 当前是第几条move语句, 用于从gradsVar中取求导变量

moveRight: 当前扫描到的节点是否在move语句的右部

1.kernel节点

最先访问的是kernel节点, 在这里先生成已知的梯度【3-5】(也就是op->outputs[0]), 存到leftVar中

然后对所有语句进行mutate操作【7-10】

新的kernel节点中的inputs部分对应于求导后等式右端出现的所有变量 (经过mutate(stmt) 操作已经把
这些变量放入了 `std::set<std::string> ins` 中), 因此只需要根据ins重新生成一遍就可以。【12-18】

而新的kernel中的outputs部分对应于json中grad_to的部分 (经过mutate(stmt) 操作已经放入
gradsVar中了) 【20】

```

1   Group visit(Ref<const Kernel> op) override {
2       std::cout<<"mutatot kernel in"<<"\n";
3       Expr lExpr = op->outputs[0];
4       std::shared_ptr<const Var> lVar = lExpr.as<Var>();
5       leftVar = Var::make(lVar->type(), "d"+lVar->name, j2i._leftAlist,
6       lVar->shape);
7
8       std::vector<Stmt> new_stmt_list;
9       for (auto stmt : op->stmt_list) {
10           new_stmt_list.push_back(mutate(stmt));
11       }
12       std::vector<Expr> inputs;
13       for (std::string s : ins){
14           std::string aim;
15           if (s[0]=='d')
16               for (int i=1;i<s.length();++i) aim+=s[i];
17           else aim=s;
18           std::vector<Expr> noUse; noUse.clear();
19           inputs.push_back(Var::make(Type::float_scalar(32), s, noUse,
20           j2i.s2Var[aim]));

```

```

19     }
20     return Kernel::make(op->name, inputs, gradsVar, new_stmt_list, op-
>kernel_type);
21 }

```

2.move节点

在【3-8】行，先给右边的式子带上梯度值（change操作。当前的梯度就是等号左边的梯度，也就是kernel中求出来的leftVar），直接mutate，执行后就完成了梯度的向下传递，同时所有变量的梯度表达式都已经存到了 `std::map<std::string, Expr> gradForVar` 中，这里由于只有一条move语句，只要执行一次就可以得到所有的梯度表达式，后续不需要再次mutate

在json_To_IRTree中已经生成了和grad_to变量一样多的move语句，这里用moveNum来记录当前处理的是哪一条，并根据moveNum来得到new_dst节点【9-10, 17】

在【11-13】行，通过当前要求梯度的变量名就可以直接找到对应的梯度表达式，存到new_src中

在【14-16】行，通过IRvisitor对该梯度表达式遍历，就可以获得所有表达式中的变量的名字，放入 `std::set ins` 中，会在生成新的kernel的inputs时用到

```

1 Stmt visit(Ref<const Move> op) override{
2     std::cout<<"mutatot move in"<<"\n";
3     if (moveNum == 0){
4         Expr newSrc = change(op->src, leftVar);
5         moveRight=true;
6         mutate(newSrc);
7         moveRight=false;
8     }
9     std::shared_ptr<const Var> _v = gradsVar[moveNum].as<Var>();
10    Expr new_dst = Var::make(_v->type(), "d"+_v->name, _v->args, _v-
>shape);
11    Expr new_src = op->src;
12    if (gradForVar.find(_v->name) != gradForVar.end())
13        new_src = gradForVar.find(_v->name)->second;
14    IRVisitor visitor;
15    new_src.visit_expr(&visitor);
16    ins.insert(visitor._ins.begin(), visitor._ins.end());
17    moveNum+=1;
18    std::cout<<"mutatot move out"<<"\n";
19    return Move::make(new_dst, new_src, op->move_type);
20 }

```

3.Binary节点

在move节点中会对moveRight进行标记，表示当前扫描的内容是move右边的表达式。

只有当 `moveRight=true` 的时候才需要梯度的下传。

【8】获得当前的梯度，后面根据加减乘除四种情况来生成新的梯度。

比如说 $c = a/b$ ，那么根据上面的讨论， a 的梯度应该是 dc/b ，因此在【29-31】行为 $op \rightarrow a$ 赋予梯度，同时mutate向下传递

```
1  Expr visit(Ref<const Binary> op) override{
2      if (moveRight == false) {
3          Expr new_a = mutate(op->a);
4          Expr new_b = mutate(op->b);
5          return Binary::make(op->type(), op->op_type, new_a, new_b);
6      }
7      std::cout<<"mutatot Binary in"<<"\n";
8      Expr nowGrad = op->grad;
9      Expr new_a, new_b;
10     if (op->op_type == BinaryOpType::Add){
11
12         Expr newA = change(op->a, nowGrad);
13         new_a = mutate(newA);
14
15         Expr newB = change(op->b, nowGrad);
16         new_b = mutate(newB);
17     }
18     else if (op->op_type == BinaryOpType::Mul){
19         Expr gradA = Binary::make(op->type(), BinaryOpType::Mul, nowGrad,
20 op->b);
21         Expr newA = change(op->a, gradA);
22         new_a = mutate(newA);
23
24         Expr gradB = Binary::make(op->type(), BinaryOpType::Mul, op-
25 >a,nowGrad);
26         Expr newB = change(op->b, gradB);
27         new_b = mutate(newB);
28     }
29     else if (op->op_type == BinaryOpType::Div) {
30
31         Expr gradA = Binary::make(op->type(), BinaryOpType::Div, nowGrad,
32 op->b);
33         Expr newA = change(op->a, gradA);
34         new_a = mutate(newA);
35
36         Expr minusA = Unary::make(op->a->type(), UnaryOpType::Neg, op->a);
37         Expr bMulb = Binary::make(op->b->type(), BinaryOpType::Mul, op->b,
38 op->b);
39         Expr gradB = Binary::make(op->type(), BinaryOpType::Mul, nowGrad,
40 Binary::make(op->type(), BinaryOpType::Div,
41 minusA, bMulb));
42         Expr newB = change(op->b, gradB);
43         new_b = mutate(newB);
44     }
45     else if (op->op_type == BinaryOpType::Sub){
46         Expr newA = change(op->a, nowGrad);
```

```

42     new_a = mutate(newA);
43
44     Expr newB = change(op->b, Unary::make(nowGrad->type(),
UnaryOpType::Neg, nowGrad));
45     new_b = mutate(newB);
46 }
47 std::cout<<"mutatot Binary out"<<"\n";
48 return Binary::make(op->type(), op->op_type, new_a, new_b);
49 }

```

4.Var节点

var节点是IRTree上的叶子部分，此时梯度已经传递到最底层了。在这个节点需要

1.把算出来的梯度表达式统计到对应的变量上去 【3-5】

这里先看一下是否已经有梯度表达式生成，如果没有则直接赋值 【4】；如果有就把当前的表达式和已经生成的求和再放回求 【5】

2.获得json中grad_to内变量的信息（下标信息） 【6-15】

在 【7-11】 先确认在gradsVar中还没有放入对应的变量的信息，然后才能把对应的信息放到 `std::vector<Expr> gradsVar` 中（这个变量在move节点会用于确认op->dst）

```

1  Expr visit(Ref<const Var> op) override {
2      std::cout<<"mutatot var in"<<"\n";
3      if (gradForVar.find(op->name) == gradForVar.end())
4          gradForVar[op->name] = op->grad; else
5          gradForVar[op->name] = Binary::make(op->type(), BinaryOpType::Add,
gradForVar[op->name], op->grad);
6      for (int i=0; i<j2i.grads.size(); ++i) {
7          bool isIn = false;
8          for (Expr e : gradsVar){
9              std::shared_ptr<const Var> _v = e.as<Var>();
10             if (_v->name == op->name) isIn=true;
11         }
12         if (op->name == j2i.grads[i] && isIn == false) {
13             gradsVar.push_back(*new Expr(op));
14         }
15     }
16     std::cout<<"mutatot var out"<<"\n";
17     return op;
18 }

```

5.其他函数

change函数用于给o表达式带上梯度（只有变量和Binary会带上梯度）

```

1  Expr change(const Expr o, const Expr nowGrad){
2      std::shared_ptr<const Binary> b = o.as<Binary>();
3      std::shared_ptr<const Var> v = o.as<Var>();
4      Expr _ret = o;
5      if (b != nullptr)
6          _ret = Binary::make(b->type(), b->op_type, b->a, b->b, nowGrad);
7      if (v != nullptr)
8          _ret = Var::make(v->type(), v->name, v->args, v->shape, nowGrad);
9      return _ret;
10 }

```

第二部分

重构后的树使用第一次project中写好的IRPrinter进行翻译与输出，只需稍加修改即可。对于IRPrinter的实现，见同文件夹下 编译技术Project1 报告。

具体例子

在这里我们以grad_case1为例。

grad_case1中式为 $C[i, j] = A[i, j] \times B[i, j] + 1.0$ ，再经过第一个project的处理后，grad_case1所生成的IRTree基本结构如下

```

1  —Groups:Kernel
2  ———Stmts:LoopNest
3  —————Stmts:Move
4  —————Expr:Binary:Add
5  —————Expr:Binary:Mul
6  —————Expr:Var:A
7  —————Expr:Var:B
8  —————Expr:FloatImm:1.0

```

根据我们的算法，算法会从 Groups 节点出发，往下搜索，当访问到 Expr:Binary:Add 节点时，此时两个子节点分别为 Expr:Binary:Mul 与 Expr:FloatImm:1.0。Expr:Binary:Add 向其子节点 Expr:Binary:Mul 与 Expr:FloatImm:1.0 传递 grad。在这里，我们规定在 Expr 中，只有对于 Expr:Binary 以及 Expr:Var 会接收父亲节点的梯度，所以对于 Expr:FloatImm:1.0 不获得梯度。而 Expr:Binary:Mul 获得梯度，根据递归公式，Expr:Binary:Mul 梯度为 grad。Expr:Binary:Mul 会继续向下传递梯度给其子节点 Expr:Var:A 和 Expr:Var:B，根据递归公式 $dA = grad \times B$ ， $dB = A \times grad$ 。此时到达叶节点，由于是对A求导，所以选择 $dA = grad \times B$ 。最后在回到 move 节点的时候，修改 move 节点的表达式变量，此时整棵语法树被重构为 $dA = grad \times B$ 。根据数学推导 $dA = dI/dC \times dC/dA = grad \times B$ ，相同，表明正确。

在输出环节，我们需要更改 Groups 中的 inputs 和 outputs 参数，outputs 参数可以通过json文件中的 grad_to 确定，得到为A，由于是求导，故在输出时，将其变量名改为dA。inputs 参数是在遍历 IRTree的时候得到确认，我们使用 set 结构来保存每个除了A的 var 节点的变量名。

最后通过命令 `Kernel::make(op->name, inputs, gradsVar, new_stmt_list, op->kernel_type);` 重新生成一棵树，并放入第一次projcet中写好的IRPrinter即可，得到正确结果。

编译知识

在本Project中，我们主要运用了编译课程中学到的抽象语法树。在Project1中，我们针对输入的json文件，通过两遍扫描，第一遍关注变量信息，第二遍生成表达式，以此实现了语法树的构建；而输出使用到了代码生成。在Project2中，第一部分中我们使用到了语法树的遍历，并运用了SDT，用于重构语法树；第二部分依然使用到了代码生成。

分工

第一部分：吴裕铖 张可鸣 王思翰 李宸昊

第二部分：李宸昊 张可鸣

接口：王思翰 张可鸣